P- 32

On-Line Replacement of Program Modules Using AdaPT

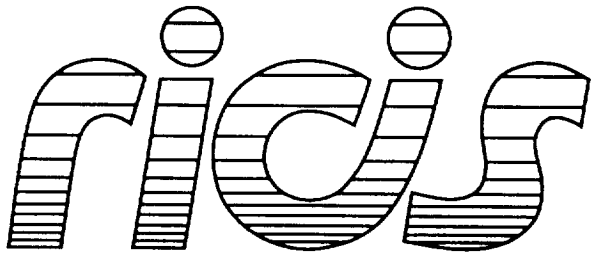
Raymond S. Waldrop
Richard A. Volz
Gary W. Smith
Texas A&M University

A.A. Holzbacher-Valero
Stephen J. Goldsack
Imperial College of London

June 5, 1992

Cooperative Agreement NCC 9-16
Research Activity No. SE.35

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Raymond S. Waldrop, Richard A. Volz, and Gary W. Smith of Texas A&M University and A. A. Holzbacher-Valero and Stephen J. Goldsack of Imperial College, London, England. Dr. E.T. Dickerson served as RICIS research coordinator.

Funding was provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Terry D. Humphrey of the Systems Software Section, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of UHCL, RICIS, NASA or the United States Government.

On-Line Replacement of Program Modules Using AdaPT

Raymond S. Waldrop, Richard A. Volz,
Gary W. Smith
Texas A&M University
A. A. Holzbacher-Valero and S. J. Goldsack,
Imperial College, London

June 5, 1992

Contents

1 Introduction

One purpose of our research is the investigation of the effectiveness and expressiveness of AdaPT[1], a set of language extensions to Ada 83, for distributed systems. As a part of that effort, we are now investigating the subject of replacing, e.g. upgrading, software modules while the software system remains in operation. The AdaPT language extensions provide a good basis for this investigation for several reasons:

- they include the concept of specific, self-contained program modules which can be manipulated,
- support for program configuration is included in the language, and
- although the discussion will be in terms of the AdaPT language, the AdaPT to Ada 83 conversion methodology being developed as another part of this project will provide a basis for the application of our findings to Ada 83 systems.

The purpose of this investigation is to explore the basic mechanisms of the replacement process. With this purpose in mind, we will avoid including issues whose presence would obscure these basic mechanisms by introducing additional, unrelated concerns. Thus, while replacement in the presence of real-time deadlines, heterogeneous systems, and unreliable networks is certainly a topic of interest, we will first gain an understanding of the basic processes in the absence of such concerns. The extension of the replacement process to more complex situations can be made later.

This report will establish an overview of the on-line upgrade problem, and present a taxonomy of the various aspects of the replacement process. Future reports will discuss specific ways AdaPT can be used to address the problem.

2 Overview of AdaPT

This section will provide a brief introduction to AdaPT. First, we will provide an overview of the new constructs introduced in AdaPT. We will then present a small example to illustrate the usage of these constructs in an AdaPT program.

2.1 Features Introduced in AdaPT

The main features introduced in AdaPT are the **partition**, the **node**, and the **public**. These are summarized below and defined in detail in [1].

- **Partitions** A partition may be considered to constitute a "class" in the sense used in object oriented systems and languages. However, it is closely modeled on the Ada package, presenting, in an interface specification, the items which are made available for its

```
partition P is
   :
end P;

partition body P is
   :
begin
   :
end P;
```

Figure 1: Sample Partition Declaration

interaction with other system components. Thus its interface may contain procedures and functions, task declarations, and constants and exception declarations. It may not contain any object or type declarations. An outline of a partition declaration is shown in Figure 1. To help in defining the initial configuration of a partition instance, a partition may have parameters (**in** parameters only), which are supplied by the program invoking the allocator when a new instance of the partition is created. The partition is the ***unit of distribution*** in AdaPT.

A partition is a library unit, and constitutes a type declaration. Other units may have **with** clauses to give them access to the definition in the library, and within the scope of the with clause they may declare variables of the type. However, the type is an implicit access type, and no instance of the partition is created by such a declaration. Creation of new instances of a partition are obtained by the use of **new** allocator statements, but these are permitted only in the definition of *nodes* which are described in the following paragraphs. Once a partition instance has been created, references to that instance may be circulated by using an assignment statement to copy the value of one (access) variable to another.

The use of library units "withed" by a partition leads to a special problem. Such packages may have "state", and consequently cannot be shared safely between different instances of a partition and between different partitions which may "with" the same unit. Thus, the semantics of **with** clauses for partitions are different from those for packages in a normal Ada program. All units in the transitive closure of the directed graph formed by the *with* clauses of a partition, up to but not including any public unit or any other partition, form part of the partition. These units are replicated as a whole with each replication of the partition. Each instance of a package or other object included in such a dependency graph, belongs therefore to one and only one partition instance. In contrast therefore to the public units described below, we sometimes refer to such packages as *non-public*

```
node N is
   ⋮
end N;

with P;
node body N is
   MY_P : P := new P'PARTITION;
   ⋮
begin
   ⋮
end N;
```

Figure 2: Sample Node Declaration

units.[1]

- **Nodes** Nodes differ very little from partitions. They too have features corresponding to those of packages; like partitions they have separate interfaces and bodies, and instance variables to reference them. However, nodes *can* create new instances of **partitions** and other nodes. Their role is to serve as units which will eventually be compiled and linked to form executable binary objects. They are thus the *units of configuration* in AdaPT. Figure 2 shows a simple node definition which includes the creation of a partition instance. Note again that MY_P is an access type which points to an instance of **partition** P. Thus, to create the object to which MY_P points, we must create an object of the anonymous type for **partition** P. This is accomplished using the attribute 'PARTITION.

  The issue of system construction and start up and elaboration is described in AdaPT as a normal Ada main program call for a first selected node, called the *distinguished* node; this then "creates" others and so recursively until the whole system is elaborated.

- **Conformant partitions** To support the provision of changed modes in a program, particularly as a technique for recovery following failure of part of the system, partitions can have "peers" which have *identical* interfaces but different bodies. In object oriented terminology they would be of the same "type", possibly one a subtype of the other, capable of providing the same set of actions for a client, albeit with different effect. In AdaPT, a conformant partition has the same interface as the partition to which it conforms, and access variables pointing to instances of conformant partitions may be used interchangeably with access variables pointing to instances of the original partition. In exact analogy with the idea of conformant partitions, it is proposed to support conformant nodes. An

[1] The word *private* has, of course, other connotations in Ada, including AdaPT.

```
partition P is
   ⋮
end P;

partition body P is
   ⋮
end P;

partition Q is P; -- Q has the same interface as P

partition body Q is -- Q may have a different body as well as different context clauses
   ⋮
begin
   ⋮
end Q;
```

Figure 3: Conformant Partition Declaration

example of the creation of a conformant partition is shown in Figure 3.

It is likely that conformant partitions may give rise to extra overhead. Some provisions may need to be made to explicitly note partitions which will certainly not have conformant peers.

- **Public Units** Partitions are permitted to share information, especially type information to give the types of the messages which form the parameters of sub-programs and task entries in the interfaces of partitions and nodes. Such sharing is permitted by sharing constant state packages. In order to enable the compiler to check that the "constant state" requirement is correctly followed, such shared units are called ***publics***. Types in public units may be private, and may be defined along with operations on them so that they are "abstract data types". Public unit interfaces may include types (except for access types), task types, task access types, static constants, subprograms (including generic subprograms), packages (which inherit the restrictions placed on public units), privates, exceptions, renames, and pragmas. The context clause of a public unit may include only other public units. An example of a public declaration is shown in Figure 4.

2.2 A Simple Example of AdaPT Usage

In this section we present a short example of how the AdaPT constructs we have presented may be used to construct an AdaPT program. In this example, a public, SENSOR_DEFS, two parti-

```
public P is
   ⋮
end P;

public body P is
   ⋮
end P;
```

Figure 4: Public Declaration

tions, SENSOR and MONITOR, and a node, N, are defined. The public provides the definition for the type used by the partitions. The SENSOR partition provides a function, SAMPLE, which returns information from a sensing device connected to the physical machine on which that instance of SAMPLE resides. The MONITOR partition contains a task, CONTROLLER, which is responsible for periodically obtaining a sample value from an instance of the SENSOR partition. MONITOR also contains a procedure, CHANGE_SENSOR, which is used to change the value of the access pointer used by CONTROLLER to indicate which instance of the SENSOR partition should be used to obtain data via the SAMPLE function. Upon its instantiation, an instance of N initially receives pointers to two instances of the SENSOR partition. These instances might reside on different physical machines, and one might be intended to serve as a backup if the other fails. The instance of N also contains a task named SWITCHER which serves as the on-going thread of control for node N[2]. Inside the task SWITCHER, we show how the access variable CURRENT_SENSOR may be set to reference different instances of the SENSOR partition type at different times.

```
public SENSOR_DEFS is
    type SAMPLE_TYPE is ...;
end SENSOR_DEFS;


with SENSOR_DEFS;
partition SENSOR is
    function SAMPLE return SENSOR_DEFS.SAMPLE_TYPE;
end SENSOR;
partition body SENSOR is
    function SAMPLE return SENSOR_DEFS.SAMPLE_TYPE is...;
end SENSOR;
```

[2]Of course, as in the case of a package in Ada, there is a separate thread of control which executes the initialization section of N. This thread of control disappears when the initialization section of N is completed.

```
with SENSOR_DEFS;
with SENSOR;
partition MONITOR is
   procedure CHANGE_SENSOR(NEW_SENSOR : in SENSOR);
   task CONTROLLER is
      entry START;
       ⋮
   end CONTROLLER;
    ⋮
end MONITOR;
partition body MONITOR is

   CURRENT_SENSOR : SENSOR;
   SAMPLE_VALUE : SENSOR_DEFS.SAMPLE_TYPE;

   procedure CHANGE_SENSOR(NEW_SENSOR : in SENSOR) is
   begin
      CURRENT_SENSOR := NEW_SENSOR;
   end;

   task body CONTROLLER is
   begin
      accept START;
      loop
         SAMPLE_VALUE := CURRENT_SENSOR.SAMPLE;
         ⋮
      end loop;
   end CONTROLLER;
end MONITOR;   -- partition


with SENSOR;
with MONITOR;
node N (SENSOR_1 : in SENSOR; SENSOR_2 : in SENSOR) is
   ⋮
end N;

node body N (SENSOR_1 : in SENSOR; SENSOR_2 : in SENSOR) is
   MY_MONITOR : MONITOR := new MONITOR'PARTITION;
   task SWITCHER is
       ⋮
   end SWITCHER;
   task body SWITCHER is
```

```
    begin
        loop
            -- wait for some signal to switch
            if CURRENT_SENSOR = SENSOR_1 then
                MY_MONITOR.CHANGE_SENSOR(SENSOR_2);
            else
                MY_MONITOR.CHANGE_SENSOR(SENSOR_1);
            end if;
        end loop;
    end SWITCHER;
    :
begin
    MY_MONITOR.CHANGE_SENSOR(SENSOR_1);
    MY_MONITOR.CONTROLLER.START;
end N;
```

To avoid overcomplicating this example, we did not give the definitions of the node or nodes which actually create these SENSOR partition instances, nor did we define a distinguished node as required in an AdaPT program. While these omissions caused the example to be incomplete, presenting the example in this form makes it easier to illustrate how the AdaPT constructs may be used in a program. Additionally, this example was intended only to give a feel for the way these constructs may be used. We therefore did not address some issues which would need to be addressed in a more complete example, such as synchronization. For example, there is a need to arrange appropriate synchronization of the use of the CHANGE_SENSOR operation and the access to the current sensor by the task controller. We will address this problem in section 5.

Having provided an introduction to AdaPT, we now discuss how AdaPT may be used to provide for on-line replacement of program modules.

3 AdaPT and the Replacement Process

AdaPT was designed to provide language support in the areas of program distribution and configuration. The features of AdaPT which provide this support are the new program unit constructs (**publics**, **partitions**, and **nodes**) and the use of the access variable paradigm as a means of referring to specific instances of partitions and nodes. The first subsection below will discuss the usefulness of these constructs in providing for replacement of program modules while the program remains on-line.

In order to achieve on-line upgrades, support for dynamic allocation and deallocation of program modules is necessary and will be discussed in separate subsections below.

3.1 AdaPT's Support For Program Configuration

The strength of AdaPT's support for program configuration lies in its explicit definition of typed modules for program distribution (partitions) and configuration (nodes), and in its use of access variables to refer to instances of those typed program modules. Because partition and node instances are instances of a type, they may be manipulated by the program itself at runtime. Instances of these types may be created in an orderly manner using the allocator, and a single access variable may be made to refer to different instances of a partition by changing the value of that access variable[3]. Program reconfiguration can thus be accomplished by changing the values of a set of access variables in an orderly manner.

3.2 Operation of the Allocator

The original definition of AdaPT merely stated that instances of partitions and nodes were created by the use of an allocator. This allocator was responsible for performing all the necessary steps for creating and initializing the unit being created. The allocator then returned a pointer to the unit thus created. No more detailed mention was made as to the means by which unit instantiation was accomplished.

To provide the capability of an on-line upgrade of a program module, it is necessary for an executing AdaPT program to be able to dynamically link with and load object code which was not in existence when the program's execution was first initiated. To provide executing AdaPT programs with this ability, we interpret the definition of the allocator to be such that it causes the underlying AdaPT run-time system to search the program library for the most recent version of the object code corresponding to the program unit of which the allocator is creating an instance. This object code will then be loaded onto the physical processor, and elaboration of the program unit instance will proceed according to the rules set forth in [1].

The dynamic linking and loading of program module instances, as described in the previous paragraph, is conceptually similar to the notion of conformant units already present in AdaPT. Both concepts involve providing multiple body implementations for a single specification, and both involve the definition of subtypes. However, the concept of conformant units is more controlled because it explicitly creates new subtypes and provides for the new subtypes thus created to be statically named at compile time. The dynamically linked and loaded subtypes created by the allocator are implicitly created.

To illustrate the dynamic creation of partition instances, we can modify the AdaPT usage example in section 2.2. Since these modifications involve only the means by which the SENSOR partition instances are created, only the configuration level of the example, i.e. node N, needs to be altered.

[3] It should be remembered that although an access variable may refer to an instance of a partition or node, that instance's existence is not dependent on that access variable. Thus, multiple access variables may refer to a single instance. However, if a situation occurs in which no access variable refers to an instance of a partition or node, there is no mechanism for rediscovering that instance, and that instance is lost to the program.

Recall that in our previous example, node N received pointers to two instances of the SENSOR partition, and switched from one to the other upon receiving a signal to do so. In this example, node N creates a new instance of the SENSOR partition upon receiving a signal to perform a dynamic replacement of the original partition instance. This occurs in the task N.SWITCHER. It should be noted that, like the previous example, this example is intended only to show the general idea of how such a dynamic replacement might be accomplished, and thus does not address all the issues associated with this replacement. These issues are, however, addressed in the example in section 5.

```
with SENSOR; use SENSOR;
with MONITOR; use MONITOR;
node N is
   ⋮
end N;

node body N is
   MY_MONITOR : MONITOR := new MONITOR'PARTITION;
   MY_SENSOR, NEW_SENSOR : SENSOR;
   ⋮
   task SWITCHER is
      ⋮
   end SWITCHER;
   task body SWITCHER is
   begin
      -- Receive the signal to perform a dynamic replacement of the sensor
      -- partition.
      NEW_SENSOR := new SENSOR'PARTITION;
      MY_MONITOR.CHANGE_SENSOR(NEW_SENSOR);
   end SWITCHER;
begin
   MY_SENSOR := new SENSOR'PARTITION;
   MY_MONITOR.CHANGE_SENSOR(MY_SENSOR);
   MY_MONITOR.CONTROLLER.START;
end N;
```

3.3 Deallocation of Partition Storage

As was mentioned above, the original definition of AdaPT in [1] made the implicit assumption that, once in use, partition instances are never discarded. Thus, no method for deallocating partition instances was discussed. There are several outstanding issues associated with such deallocation which merit further study. In this report, we will not address those issues. However,

to provide a flavor of the possible use of such deallocation, we make use of a variant of Ada 83's UNCHECKED_DEALLOCATION procedure.[4] The procedure we will use has this form:

```
generic
   type NAME is partition;
procedure UNCHECKED_DEALLOCATION(X : in out NAME);
```

An instantiation of this generic procedure is made using the name of the partition type that will be deallocated. (Recall that a partition declaration defines an access type to an anonymous type.) An example of such an instantiation follows:

```
partition SENSOR is
   ⋮
end SENSOR;

procedure UNCHECKED_SENSOR_DEALLOCATION is new
                    UNCHECKED_DEALLOCATION(SENSOR);
```

To illustrate the usage of this deallocator, consider the following example where we have modified the node N from the previous example to use UNCHECKED_DEALLOCATION. Notice that the deallocation of the old SENSOR partition allows us to create new partitions as needed within a loop.

```
node body N is
   MY_MONITOR : MONITOR := new MONITOR'PARTITION;
   MY_SENSOR, NEW_SENSOR : SENSOR;
   procedure UNCHECKED_SENSOR_DEALLOCATION is new
                    UNCHECKED_DEALLOCATION(SENSOR);
   ⋮
   task SWITCHER is
      ⋮
   end SWITCHER;
   task body SWITCHER is
   begin
      loop
         -- Receive the signal to perform a dynamic replacement of the sensor
         -- partition.
         NEW_SENSOR := new SENSOR'PARTITION;
         MY_MONITOR.CHANGE_SENSOR(NEW_SENSOR);
         UNCHECKED_SENSOR_DEALLOCATION(MY_SENSOR);
```

[4] See [2] for additional discussion of this issue.

```
            MY_SENSOR := NEW_SENSOR;
        end loop;
    end SWITCHER;
begin
    MY_SENSOR := new SENSOR'PARTITION;
    MY_MONITOR.CHANGE_SENSOR(MY_SENSOR);
    MY_MONITOR.CONTROLLER.START;
end N;
```

A potential problem exists if UNCHECKED_SENSOR_DEALLOCATION(MY_SENSOR) is called when MY_SENSOR is a remote partition. In this case, an exception may need to be raised, but further study is needed. Also, the user is expected to have aborted any active tasks within the partition before using the deallocator.

4 The Replacement Process

Having presented an overview of AdaPT, we now begin a discussion of the replacement process itself. First, we discuss five parameters which may be used to characterize the replacement process. We then use these parameters to form a taxonomy of the replacement process.

4.1 Characterization of the Replacement Process

The complexity of the general problem of program module replacement is due to the wide variety of situations under which the replacement process must occur. The study of this problem can be simplified by breaking it down into a number of cases. To allow the problem space to be broken down, we have determined five parameters which can be used to classify instances of the problem. These parameters are:

- the type of replacement,
- the type of module to be replaced,
- the location of the replacement module(s),
- the need for a replacement module's state to match that of the module it is replacing, and
- the degree of change involved between the specification of the original module and the specification of its replacement.

These five parameters will be explained below.

Types of Replacement

We divide replacement processes into two types: *planned* and *unplanned*. A planned replacement is one where the system knows about the upcoming replacement before the module to be replaced is deactivated. An unplanned replacement is one where the system does not know of the need for the replacement until the module in question is found to be no longer in service. The main difference between these two cases is that the system designer typically has more options open to him in the planned case due to the fact that the original module is still available for use. An example of a planned replacement is that of an operator instructing the system to replace a program module with a new version of that module. (This new version would presumably incorporate bug fixes, expanded capabilities, etc.) An example of an unplanned replacement is that of a loss of power to a physical machine. The latter would result in the unexpected loss of all system functions resident on that processor.

Kinds of Replacement Modules

The design of AdaPT provides two syntactic units that can be replaced, nodes and partitions. These are the only module types whose replacement will be considered in this discussion. The replacement of a node will usually require the replacement of its partitions.

Possible Locations

There are three possible situations regarding the location of the replacement module(s):

- *local*, meaning the replacement module is to reside on the same node as the module being replaced,
- *remote*, meaning the replacement module is to reside on a different node from the module being replaced, and
- *multiple remote*, meaning that various portions of a node are replaced by modules on different nodes.

State

There are significant differences in the replacement process depending upon the role of state in the module being replaced. There are two different cases to be considered:

- The module to be replaced has no state and creates no state via the allocator.
- The module to be replaced contains state whose consistency must be maintained throughout the replacement process.

Module Specifications

As a program evolves over time, changes will be made to various modules of that program. These changes fall into three categories:

- changes in which the module's specification remains unchanged, as in conformant partitions in AdaPT,
- changes in which the module's specification is extended, i.e. items are added to the module's specification, as in inheritance in object oriented languages, and
- changes in which the module's specification is reduced, i.e. items are removed from the module's specification, as is permitted in some object oriented laguages.

A fourth category, where some items are added to the module's specification while other items are removed, is merely a composition of the second and third categories listed above. We will therefore not address this fourth category separately.

4.2 Taxonomy of the Replacement Process

In the previous section, we presented five parameters of the replacement process. In our discussions of these parameters, we listed the possible values these parameters may take on. Any instance where a module is to be replaced may be classified by listing the values of these parameters. Because there are a finite number of parameters, and a finite number of values for those parameters, there is a finite number of combinations of those parameters values. Additionally, some combinations of parameter values will not occur.

To aid in understanding what cases are possible, we have created the acyclic directed graph shown in Figure 5. In this graph, the vertices other than "Enter" and "Exit" represent the possible values for the five parameters of the replacement process, with the vertices representing values corresponding to the same parameter being placed at the same level as measured from the "Enter" vertex. The arcs connecting the nodes represent possible combinations, i.e. the presence of an arc from "Partition" to "Remote" indicates that this combination of values is permissible, while the absence of an arc from "Partition" to "Multiple Remote" indicates that this combination is not permissible. A path describing a module replacement situation may be obtained by traversing the graph from vertex marked "Enter" to the vertex marked "Exit". At each vertex encountered during the traversal, the path should follow the arc to the vertex which represents the parameter value corresponding to the situation being classified, or the arc to "Exit" in the case of the last level of vertices.

Our approach to the replacement problem is to investigate the various possible cases to learn what techniques are needed to solve that particular case. These techniques can then be applied to solve the general case. In the next section, we present a solution to one of the possible module replacement situations.

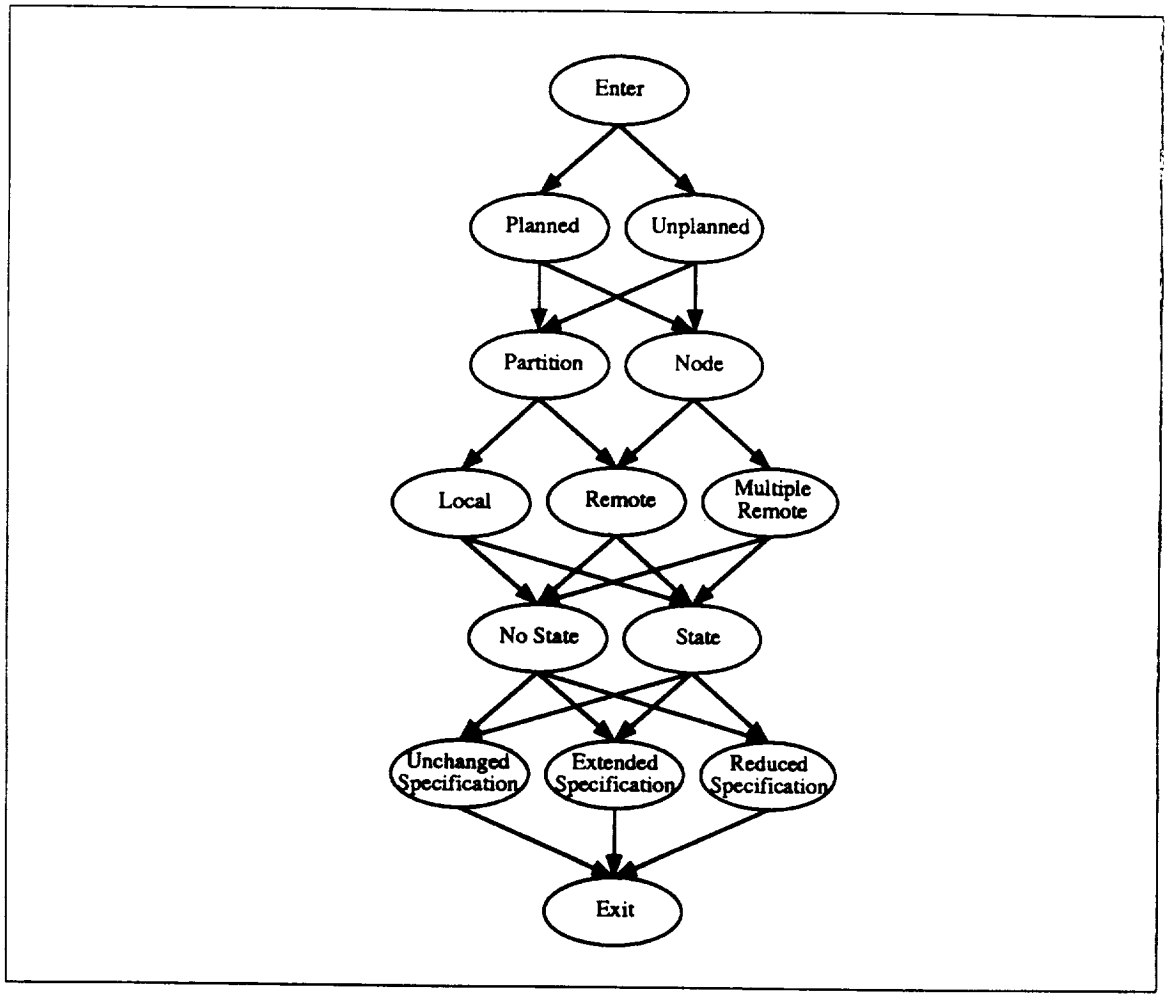


Figure 5: Reconfiguration Situation Classification Graph

5 Replacement Example

In this section, we will provide a simple example of the replacement process. First, we will describe the problem and classify it according to the parameters listed above. Next, we will show a simple implementation of the example in AdaPT.

5.1 Example Description

As was stated earlier, our intention is to study the replacement process itself and to avoid the inclusion of concerns not directly related to the basic replacement process. With this in mind, we have chosen a simple example to illustrate a basic replacement process. In this example, we have constructed a node which is responsible for maintaining a server partition. This server partition is to be used by clients on remote nodes. The node we have constructed also has the capability to replace the server partition upon receiving an appropriate signal from some remote entity.

When replacing a partition instance, the instance being replaced must not be deallocated until all potential clients have been notified of the new partition instance. In our solution we require that all potential clients must "register" with the node before using the server. As clients make calls to the server partition instance which is to be replaced they are notified of the change and given access to the new server partition instance. The node keeps track of how many clients have been notified of the change. When all have been notified, the old server partition instance can be deallocated.

It should be noted that the maximum amount of time that will transpire before the old server partition instance can be deallocated is the potential maximum amount of time between server calls for any client which has registered. In the case of periodic clients with long periods or for aperiodic clients, this wait may be unduely long. For these cases, a "de-register" operation is provided. After de-registering, the clients must register again before using the server.

This solution assumes that the clients will not send a second request until they have received a reply to their first request. If the clients did not wait for the reply to their request, the node would have to employ some other mechanism to determine whether it could safely deallocate the partition instance being replaced. To avoid obscuring the objective of our example, i.e. to study the underlying mechanisms of one case of the replacement problem, we chose not to include such considerations in this example.

5.2 Example Classification

This small example can be classified simply. Traversing the replacement classification graph in Figure 5, we classify this example as an instance of planned, partition, local, no state, unchanged specification replacement. The classification path corresponding to this example is shown in

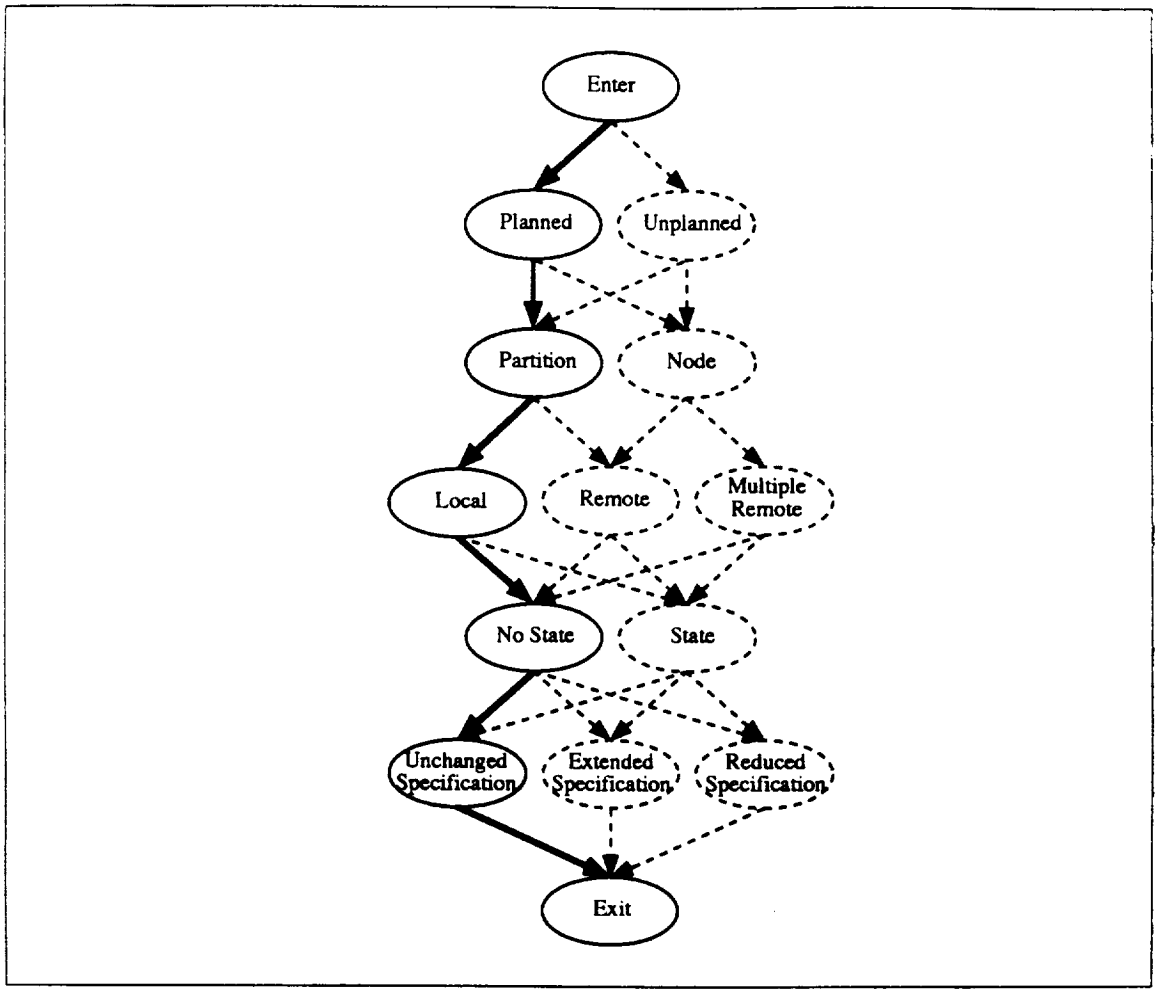


Figure 6: Replacement Example Classification

Figure 6. In this figure, nodes and edges on the path are dark, while those not on the path are dotted.

5.3 Example Implementation

In our implementation of this example problem, we define four program units:

- **package** LOCKER,
- **public** COORDINATOR_DEFINITION,
- **partition** SERVER_TYPE, and

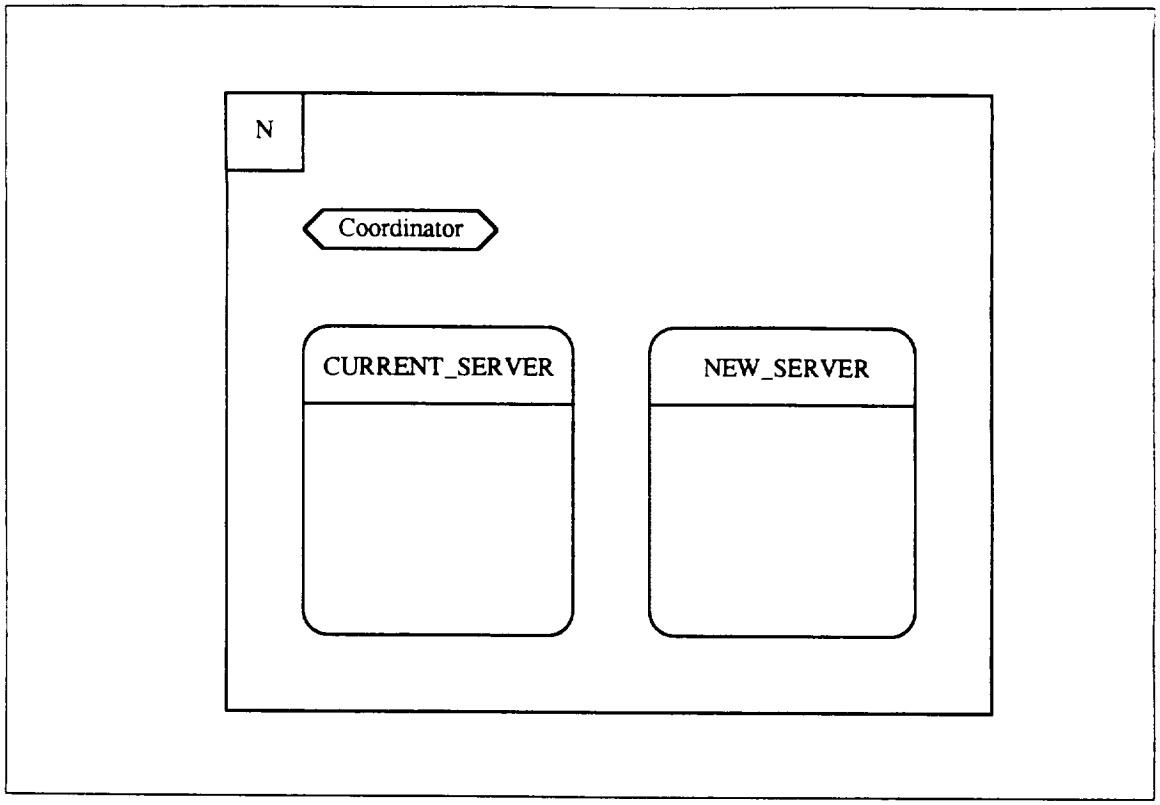


Figure 7: Diagram of Example Implementation

- **node** N.

A problem could occur if an access variable pointing to a partition instance could be changed by one thread of control while the access variable is being used by another thread of control. This problem corresponds to the Readers-Writers problem and in our example, the package LOCKER is used to solve it.

Public COORDINATOR_DEFINITION provides a task type which is used by instances of partition SERVER_TYPE and node N to coordinate the replacement process.

Partition SERVER_TYPE implements the service provided to the clients.

Node N creates instances of partition SERVER_TYPE as needed, and provides overall control of the replacement process.

A simple diagram of node N, which consists of the coordinator partition and server partitions, is shown in Figure 7. The following is our implementation of this example:

```
-- This package provides a simple solution to the readers-writers problem.
-- It is modelled after the solution in Barnes' "Programming in Ada", 3rd ed.
```

```
package LOCKER

   task type LOCK is
      entry READ;
      entry WRITE;
      entry DONE;
   end LOCK;

end LOCKER;

package body LOCKER is

   task body LOCK is
      NO_WRITE : BOOLEAN := FALSE;
      READERS : NATURAL := 0;
   begin
      accept WRITE;
      accept DONE;
      CONTROL:
      loop
         select
            accept READ;
            READERS := READERS + 1;
         or
            accept DONE;
            READERS := READERS - 1;
         or
            accept WRITE do
               CLEAR_READERS:
               while READERS > 0 loop
                  accept DONE;
                  READERS := READERS - 1;
               end loop CLEAR_READERS;
            end WRITE;
            accept DONE;
         end select;
      end loop CONTROL;
   end LOCK;

end LOCKER;

-------------------------------------------------------------

-- This public provides a tasked used for communication between the server
-- partition and its controlling node during the replacement process.
public COORDINATOR_DEFINITION is

   task type COORDINATOR_TYPE is
```

```
      entry ADD_CLIENT        -- A new potential client has 'registered'.
      entry CHANGE_MADE;      -- A client has 'checked in' - ie: a previously
                              --    register client has been notified of the
                              --    server change.
      entry RESET_COUNT;      -- Reset the count of clients who have checked in.
      entry COUNT_REACHED;    -- Accepted only when all clients have checked in.
      entry DELETE_CLIENT;    -- A potential client has 'deregistered' either
                              --    when there is not a server change in progress
                              --    or when there is a server change but the
                              --    client wasn't aware of the change (change
                              --    made wasn't called for it.)
      entry DELETE_UPDATED_CLIENT;
                              -- A potential client has 'deregistered' during
                              --    a server change and the client had previously
                              --    been notified of the server change (change
                              --    made was previously called for it.)
   end COORDINATOR_TYPE;

end COORDINATOR_DEFINITION;

public body COORDINATOR_DEFINITION is

   task body COORDINATOR_TYPE is
      NUM_CLIENTS : NATURAL := 0;
      NUM_CHANGED : NATURAL := 0;
   begin
      loop
         select
            accept ADD_CLIENT;
            NUM_CLIENTS := NUM_CLIENTS + 1;
         or
            accept CHANGE_MADE;
            NUM_CHANGED := NUM_CHANGED + 1;
         or
            accept RESET_COUNT;
            NUM_CHANGED := 0;
         or
            when NUM_CHANGED = NUM_CLIENTS =>
            accept COUNT_REACHED;
         or
            accept DELETE_CLIENT;
            NUM_CLIENTS := NUM_CLIENTS - 1;
         or
            accept DELETE_UPDATED_CLIENT;
            NUM_CLIENTS := NUM_CLIENTS - 1;
            NUM_CHANGED := NUM_CHANGED - 1;
         or
            terminate;
```

```
            end select
        end loop;
    end COORDINATOR_TYPE;

end COORDINATOR_DEFINITION;

-------------------------------------------------------------

-- This partition defines a server.  While the service it performs is quite
-- simple, it is sufficient for the purpose of this example.
with COORDINATOR_DEFINITION;
partition SERVER_TYPE(COORDINATOR : in
                                    COORDINATOR_DEFINITION.COORDINATOR_TYPE) is

    -- This is the service procedure provided by this server.  The X parameter
    -- is the only parameter used to perform this service.  The second parameter,
    -- CALL_NEXT, is used to redirect the client to the new server once a
    -- replacement process has been initiated.

    procedure P(X : in out INTEGER; CALL_NEXT : in out SERVER_TYPE);

    -- This procedure is called by the controlling node to notify the partition
    -- that it is being replaced.  It also passes a pointer to the replacement
    -- partition, so this partition can pass it on to the clients.
    procedure SET_NEXT(NEXT : in SERVER_TYPE);

end SERVER_TYPE;

with LOCKER;
partition body SERVER_TYPE(COORDINATOR : in
                                    COORDINATOR_DEFINITION.COORDINATOR_TYPE) is

    -- This variable is used to hold a pointer to the partition instance that
    -- should be used for the next call made by the client.
    NEXT_SERVER : SERVER_TYPE;

    -- This task is used to control access to the variable NEXT_SERVER.
    NEXT_LOCK : LOCKER.LOCK;

    procedure P(X : in out INTEGER; CALL_NEXT : in out SERVER_TYPE) is
    begin

        X := X * 2; -- Just a simple function to do some manipulation on X.

        -- The remainder of this procedure deals with the replacement
        -- process.

        -- First, we must get read access to the pointer to the partition to be
```

```
        -- used for the next call.
        NEXT_LOCK.READ;

        -- If a different partition is to be used for the next call, pass back
        -- the pointer to that partition and make a note that another client
        -- has been informed of the change.
        if NEXT_SERVER <> CALL_NEXT then
            CALL_NEXT := NEXT_SERVER;
            COORDINATOR.CHANGE_MADE
        end if;

        -- Now, release the lock on the pointer to the next partition to be used.
        NEXT_LOCK.DONE;

    end P;

    -- This procedure is used to set the partition's pointer to the partition
    -- to be used for the next call.
    procedure SET_NEXT(NEXT : in SERVER_TYPE) is
    begin
        NEXT_LOCK.WRITE;
        NEXT_SERVER := NEXT;
        NEXT_LOCK.DONE;
    end SET_NEXT;

end SERVER_TYPE;

-------------------------------------------------------------------

-- This node creates server partitions and replaces them upon receiving
-- commands from an external source.
node N is

    -- This function returns the value of the current server.  To conform to the
    -- replacement protocol, a client MUST call this function BEFORE using the
    -- server.  Calling this function "registers" the client with the
    -- COORDINATOR task.  This registration is important to the replacement
    -- process.
    function REGISTER_CLIENT return SERVER_TYPE;

    -- This procedure de-registers a client.  Used when a client no longer
    -- needs the server or when there may be a long time before the next
    -- usage.  Server is needed as a paramater to note if the client
    -- has been told of possible server change in progress.
    procedure DEREGISTER_CLIENT (SERVER : in out SERVER_TYPE);

    -- This procedure causes the node to replace the active server partition
    -- instance with a partition instance of the type currently in the library.
```

```
    procedure INITIATE_REPLACEMENT;

end N;

with LOCKER;
with COORDINATOR_DEFINITION;
node body N is

    -- This task is used to coordinate the replacement activities of this
    -- node and its server partition.
    COORDINATOR : COORDINATOR_DEFINITION.COORDINATOR_TYPE;

    -- These variables are used to point to the server partitions used by this
    -- node. Note that no more than two such partitions will be in use at
    -- any given time.
    CURRENT_SERVER, NEW_SERVER : SERVER_TYPE;

    -- This flag is used to indicate that the current server partition instance
    -- is actually in the process of being replaced. This implies that the
    -- replacement server partition instance has already been created and
    -- initialized.
    SWITCHING_SERVERS : BOOLEAN := FALSE;

    -- This task is used to control access to the CURRENT_SERVER and
    -- SWITCHING_SERVERS variables.
    SERVER_LOCK : LOCKER.LOCK;

    -- This task is used to control the replacement process.
    task REPLACEMENT_CONTROL is

        -- This entry MUST be called first. This ensures that no replacement
        -- process can begin until the node and the partition are ready.
        entry START;

        -- This entry is indirectly called by a remote entity via the procedure
        -- PERFORM_REPLACEMENT. When this entry is called, a replacement
        -- process begins. No other entry calls will be accepted until the
        -- replacement is completed.
        entry PERFORM_REPLACEMENT;

    end REPLACEMENT_CONTROL;

    task body REPLACEMENT_CONTROL is
        TEMP : SERVER_TYPE;
    begin

        -- This task will not procede past this point until the node
        -- initialization section calls this entry to signal that everything
```

```
-- is ready.
accept START;

-- This is the main loop of this task.  Inside this loop, the task
-- waits for the signal to replace the server partition.  When this
-- signal comes (the entry is called), the task creates a new server
-- using the version currently in the library, waits until all known
-- clients have been notified to use the new server, and deallocates
-- the old server.
REPLACEMENT_LOOP:
loop
   select

      -- This entry signals that the server is to be replaced.
      accept PERFORM_REPLACEMENT;

      -- To start the process, tell the COORDINATOR task to start
      -- counting clients that check in and are told that the server
      -- is being replaced.
      COORDINATOR.RESET_COUNT;

      -- Create a new server partition, using the version currently
      -- in the program library.
      NEW_SERVER := new SERVER_TYPE'PARTITION(COORDINATOR);

      -- Set up the new server's pointer to the server to be used
      -- for the next call.
      NEW_SERVER.SET_NEXT(NEW_SERVER);

      -- Now that the new server partition instance has been created
      -- and initialized, set the flag SWITCHING_SERVERS to indicate
      -- that the actual switch is now taking place.
      SERVER_LOCK.WRITE;
      SWITCHING_SERVERS := TRUE;
      SERVER_LOCK.DONE;

      -- Tell the current server that clients should be told to use
      -- the new server.
      CURRENT_SERVER.SET_NEXT(NEW_SERVER);

      -- This entry call will block until all clients have been
      -- informed about the server change.
      COORDINATOR.COUNT_REACHED;

      -- At this point, all clients know to use the new server partition.
      -- This leaves us free to dispose of the old server partition.
      -- First, we save a pointer to the old partition.
      -- Second, we make the new server the current server.
```

```
            -- Third, we reset the flag SWITCHING_SERVERS to indicate that
            -- the switch has been completed.
            -- Finally, we dispose of the old server partition.  For a
            -- discussion of UNCHECKED_DEALLOCATION of partitions, please
            -- see the discussion of this example elsewhere in the report.

            -- For these operations we need to get exclusive access to the
            -- server variable CURRENT_SERVER and the flag SWITCHING_SERVERS.
            SERVER_LOCK.WRITE;
            TEMP := CURRENT_SERVER;
            CURRENT_SERVER := NEW_SERVER;
            SWITCHING_SERVERS := FALSE;
            SERVER_LOCK.DONE;

            -- The actual deallocation of the storage used by the discarded
            -- partition instance can be accomplished without locking access
            -- to the instance currently in use.
            UNCHECKED_DEALLOCATION(TEMP);
        or
            terminate;
        end select;
    end loop REPLACEMENT_LOOP;

end REPLACEMENT_CONTROL;

-- This function returns a pointer to the server currently in use.  Note
-- that this function will block while task REPLACEMENT_CONTROL is actually
-- performing a server change.
function REGISTER_CLIENT return SERVER_TYPE is

    -- This temporary variable must be used to allow us to relinquish our
    -- READ access to the CURRENT_SERVER variable before exiting.
    TEMP : SERVER_TYPE;

begin

    -- This informs the COORDINATOR task that the number of clients needs
    -- to be incremented.
    COORDINATOR.ADD_CLIENT;

    -- This function cannot pass beyond this point while another thread of
    -- control has WRITE access to the variable CURRENT_SERVER and the flag
    -- SWITCHING_SERVERS.
    SERVER_LOCK.READ;

    -- If a new server partition instance is being brought into use, return
    -- an access variable to the new instance.  Otherwise, return an access
    -- variable to the instance currently in use.
```

```
    if SWITCHING_SERVERS then
       TEMP := NEW_SERVER;

       -- This entry call must be made to inform the COORDINATOR task that
       -- another client has been notified that it should use the new sever
       -- partition instance.
       COORDINATOR.CHANGE_MADE;
    else
       TEMP := CURRENT_SERVER;
    end if;

    SERVER_LOCK.DONE;

    return TEMP;
end REGISTER_CLIENT;

-- This procedure deregisters a client.  Used when a client no longer
-- needs the server or when there may be a long time before the next
-- usage.  Server is needed as an in paramater to note if the client
-- has been told of possible server change in progress...Server is set
-- to null on the way out.
procedure DEREGISTER_CLIENT (SERVER : in out SERVER_TYPE);

begin

    -- This function cannot pass beyond this point while another thread of
    -- control has WRITE access to the variable CURRENT_SERVER and the flag
    -- SWITCHING_SERVERS.
    SERVER_LOCK.READ;

    -- If a new server partition instance is being brought into use, need
    -- to note whether this client had been told of that change.
    if SWITCHING_SERVERS and (SERVER = NEW_SERVER) then

       -- Delete (deregister) a client that had been previously told of a
       -- pending server change.
       COORDINATOR.DELETE_UPDATED_CLIENT;
    else
       -- Delete (deregister) a client that either had not been previously
       -- told of a pending server change or there is no pending server
       -- change
       COORDINATOR.DELETE_CLIENT;
    end if;

    SERVER_LOCK.DONE;

    SERVER := null;
end REGISTER_CLIENT;
```

```
    -- This procedure is used to hide task REPLACEMENT_CONTROL from the outside
    -- users.  This hiding ensures that only node N can start the
    -- REPLACEMENT_CONTROL task.
    procedure INITIATE_REPLACEMENT is
    begin
        REPLACEMENT_CONTROL.PERFORM_REPLACEMENT;
    end INITIATE_REPLACEMENT;

begin

    -- This is guaranteed to be the first call accepted by SERVER_LOCK because
    -- the first entry call accepted will be a WRITE, and because the only
    -- other source of a WRITE call is task REPLACEMENT_CONTROL which will
    -- block until it receives the START call to be sent a few lines after this.
    SERVER_LOCK.WRITE;

    -- Create a sever partition and pass it a reference to the COORDINATOR task.
    CURRENT_SERVER := new SERVER_TYPE'PARTITION(COORDINATOR);

    -- Initialize the server partition's next call pointer to the partition
    -- itself.
    CURRENT_SERVER.SET_NEXT(CURRENT_SERVER);

    -- Release the lock on the server partition.
    SERVER_LOCK.DONE;

    -- Let the REPLACEMENT_CONTROL task go into its loop and wait for a signal
    -- to perform a replacement operation.
    REPLACEMENT_CONTROL.START;

end N;
```

6 Future Work

In this work we have initiated a discussion of on-line program module upgrades. We have presented a system for classifying the various situations which arise in this problem. Also, we have presented a solution to one of these situations. We propose to continue our investigations of this problem in the following order (with reference to the taxonomy we presented):

- replacement of partitions when state must be transferred between partition instances,
- replacement of partitions where the replacement instance is located on a different node from the instance being replaced,

- replacement of a node by another node instance,
- replacement of a node where the partition replacement instances will be located on several different nodes,
- replacement of a partition with an extended specification, and
- replacement of a partition with a reduced specification.

There are two additional topics which deserve further study. The first topic is the use of UNCHECKED_DEALLOCATION to deallocate partitions. The second topic concerns the use of dynamic linking and loading by the allocator. Problems may arise in determining which version of the object code was used to instantiate a particular instance of a program unit. This is similiar to polymorphism and dynamic binding of procedure calls in object oriented languages. We will investigate these topics further as we continue our research.

References

[1] A. B. Gargaro, S. J. Goldsack, R. A. Volz, and A. J. Wellings, "A Proposal to Support Reliable Distributed Systems in Ada 9X," Technical Report 90-10, Department of Computer Science, Texas A&M University, College Station, Texas, 1990.

[2] A. A. Holzbacher-Valero, S. J. Goldsack, R. Volz and R. Waldrop. "Transforming AdaPT to Ada83," Status Report, subcontract #074 cooperative agreement NCC-9-16.